Automation of programming for promising high-performance computing systems

Vladimir Bakhtin, Dmitry Zakharov,
Nikita Kataev, Alexander Kolganov,
Mikhail Yakobovskiy

Keldysh Institute of Applied Mathematics RAS,
Moscow, Russia

22 August, 2023 | Astana, Kazakhstan

Agenda

- Today's parallel programming and introduction to the DVMH directive-based programming model.
- Typical parallelization strategy in the DVMH model from a sequential program to a tuned parallel one for heterogeneous cluster:
  - ✓ program analysis and profiling,
  - ✓ parallelization for multiprocessor, GPUs and distributed memory system,
  - ✓ program debugging and performance analysis.
- Overview of programs developed in the DVMH model with the help of System FOR Automated Parallelization (SAPFOR) on the example of the parallelization of a software packaged for numerical simulation of hydrodynamic instabilities.
- Conclusion.

Parallel programming tools

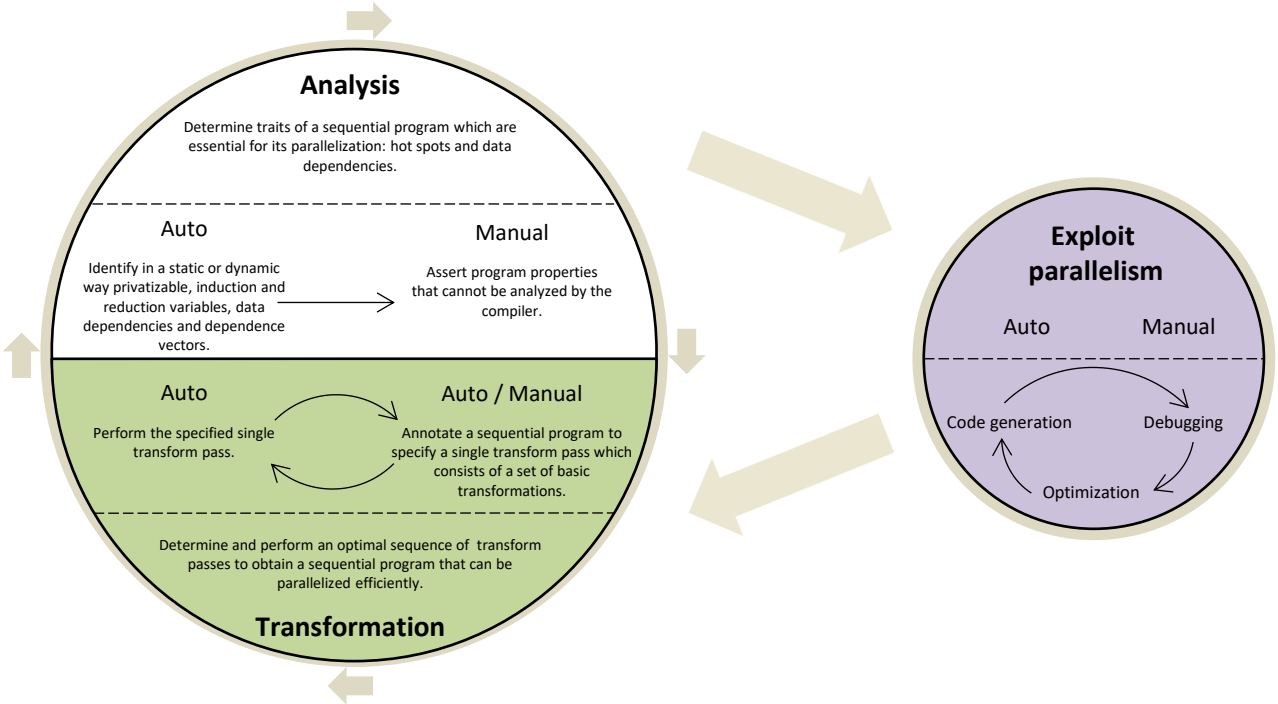
Performance → Convenience

MPI, CUDA, OpenCL, SHMEM, pThreads	OpenMP, OpenACC, DVMH, XcalableACC, Halid, Vobla, Graphit, MKL, Thrust, cuBLAS	SAPFOR, PPCG, Polly, Pluto, Apollo, Paradigm, SUPERB
Low-level models give programmers fine-grained control over the program execution and allow them to gain the best performance.	Directive-based models, DSLs and general purpose libraries simplify programming and increase software maintainability while still providing high performance.	Automatic parallelizing compilers return a fully parallelized source code (maybe not optimal) for a given sequential one.

Blended approach to parallel programming

Try to automate as much as possible, but allow the programmer to participate in parallelization if necessary.

- Parallelization process is considered as a sequence of separate steps.
- Each step can be automated if possible or, otherwise, executed manually.



Three main parts of the blended approach

High-level programming model

- General enough to solve tasks in a wide domain.
- Wide and extendable enough to support different parallel architectures.
- Implicit enough to hide from the application programmer implementation details.
- Explicit enough to allow the compiler to optimize programs for a chosen architectures.

Automation tools

- Exploration of sequential programs (program analysis and profiling).
- Automatic parallelization (according to the high-level programming model) of a well-formed program for which a programmer maximizes algorithm-level parallelism and asserts high-level properties (implicit parallel programming methodology).
- Semi-automatic program transformation to obtain a well-formed sequential version of the original program.

User participation

The DVMH programming model

DVMH

Development of high-level parallel programming languages. Directive-based programming models.

Directive-based programming model which aims to create parallel programs for heterogeneous computational clusters (GPU NVidia, Intel Xeon Phi, multicore CPUs).

The model includes two programming languages which are the extensions of standard C and Fortran languages by parallelism specifications: *CDVMH* and *Fortran-DVMH*

The parallel program is developed in terms of a sequential one.

MPI

OpenMP CUDA

Data distribution

- Distribution of array elements across computational nodes:

  *directives **distribute / align***

Computation distribution

- Mapping of the loop iterations on the processors in accordance with data distribution:

  *directive **parallel***

Variable properties and remote data

- Organization of the efficient access to remote data located on other processors:

  *clauses **shadow / across / remote***

- Organization of the efficient execution of reduction operations which are global operations on the data located on different processors:

  *clause **reduction: max/min/sum/maxloc/minloc/…***

Compute regions and specifications of CPU-to-GPU data transfer

- Specification of the regions which are special constructions of the DVMH languages. These constructions consist of sequential parts of code and parallel loops. The regions can be executed on the accelerators:

  *directive **region***

- Specification of the actualization directives which control data movement between a memory of CPU and memories of accelerators:

  *directives **actual / get_actual***

Features of the DVMH model

Parallelism exploitation

- Programming is accomplished in a sequential style.
- A normal compiler neglects specifications of parallelism, so the same program is suited for sequential and for parallel execution.
- Specifications of a low-level data transfer and synchronization are absent in a source code.
- High-level parallelism specifications preserve source code readability and maintainability.

Dynamic tuning methods

- The advanced runtime system manages the program execution and adapts it to all available resources.

- Optimizations hidden from the user:
  - ✓ data transformation at runtime to choose the right memory access pattern,
  - ✓ dynamic CUDA handler compilation during the program runtime,
  - ✓ parallel execution of loops with regular loop-carried dependences.

Debugging (dynamic analysis, comparative debugging) and performance analysis and prediction

- All analysis tools operate in terms understandable to a user.

Typical parallelization strategy in the DVMH model

1. Migration of a program to a target OS and to a target compute system, the original program debugging.
2. Program profiling.
3. Parallelization of the most time-consuming source code regions:
   - analysis of chosen code regions (exploration of control-flow and data-flow graphs, memory access pattern, loop-carried data dependencies as well as spurious dependencies),
   - incremental parallelization for shared memory (multiprocessor and GPU),
   - incremental parallelization for distributed memory to asses possible performance gain on a distributed memory system.
4. Parallelization of the entire program taking into account the already found solutions.
5. Performance analysis and search for an optimal execution configuration of a program (number of processors to use, number of dimensions in the processor grid as well size of each dimension, distribution of computations between GPUs and CPU cores inside each compute node).
6. Increasing of a program reliability and reduction of input/output overhead (insertion of checkpoints and parallel IO).

Example: solving heat equation using Jacobi iterative method

```
program jacobi
  double precision, allocatable, dimension(:,:,:) :: f, newf, r
  ...
  allocate(f(mx, my, mz))
  allocate(newf(mx, my, mz))
  allocate(r(mx, my, mz))
  curf = 0
  do n = 1, NITER
    if (curf .eq. 0) then
      eps = dostep(f, newf, r, rdx2, rdy2, rdz2, beta, mx, my, mz)
    else
      eps = dostep(newf, f, r, rdx2, rdy2, rdz2, beta, mx, my, mz)
    endif
    print *, 'Iteration=' , n, 'eps=', eps
    curf = 1 - curf
    enddo
end

double precision function dostep(f, newf, r, rdx2, rdy2, rdz2, beta, mx, my, mz)
  integer :: mx, my, mz
  double precision, dimension(mx,my,mz) :: f, newf, r
  double precision :: rdx2, rdy2, rdz2, beta, eps
  integer :: i, j, k
  eps = 0.
  do k = 2, mz - 1
    do j = 2, my - 1
            do i = 2, mx - 1
              newf(i, j, k) = ((f(i-1,j,k)+f(i+1,j,k))*rdx2+(f(i,j-1,k)+f(i,j+1,k))*rdy2
&                                                   +(f(i,j,k-1)+f(i,j,k+1))*rdz2-r(i,j,k)) * beta
              eps = max(eps,abs(newf(i,j,k)-f(i,j,k)))
            enddo
          enddo
  enddo
  dostep = eps
end function
```

Step 1: migration to the target system and debugging

Specific debugging tools are required to ensure the similar program behavior on different OS and compute systems (even in case of sequential programs).

Different compilers as well as different versions of the same compiler may affect program behavior and reveal previously hidden errors under new optimization circumstances.

```
BW: [4] "eps"; {"test.f", 26}
AW: [4] "eps" = 0; {"test.f", 26}
PL: 2() [2]; {"test.f", 29}, 96.B
  IT: 18, (2,2)
  BW: [4] "b(i,j)"; {"test.f", 31}
  RD: [4] "a(i - 1,j)" = 681.72562479972839; {"test.f", 31}
  RD: [4] "a(i + 1,j)" = 551.3431462012436; {"test.f", 31}
  RD: [4] "a(i,j - 1)" = 681.72562479972839; {"test.f", 31}
  RD: [4] "a(i,j + 1)" = 551.3431462012436; {"test.f", 31}
  AW: [4] "b(i,j)" = 616.534385500486; {"test.f", 31}
  <...>
EL: 2; {"test.f", 33}, 96.E
PL: 3() [2]; {"test.f", 36}, 97.B
  IT: 18, (2,2)
  RV_BW: [4] "eps"; {"test.f", 38}
  RD: [4] "a(i,j)" = 616.51675929759654; {"test.f", 38}
  RD: [4] "b(i,j)" = 616.534385500486; {"test.f", 38}
  RV_AW: [4] "eps" = 0.017626202889459819; {"test.f", 38}
  BW: [4] "a(i,j)"; {"test.f", 39}
  RD: [4] "b(i,j)" = 616.534385500486; {"test.f", 39}
  AW: [4] "a(i,j)" = 616.534385500486; {"test.f", 39}
  <...>
```

The DVM system provides the user with a comparative debugging tool:

- Comparative debugging relies on the accumulation of calculation results in a program execution trace with subsequently comparison of the gathered trace with a program behavior under other circumstances.
- The program trace allows the debugging tool to determine the place in the program and the moment when unexpected behavior occurs.
- The trace comprises all variable accesses, the beginning and ending of the execution of loop iterations.
- Compiler time and runtime options can be used to select events for tracing.

Step 2: sequential program profiling

```
INTERVAL ( NLINE=12 SOURCE=jacobi.f ) LEVEL=3 SEQ EXE_COUNT=5120
--- The main characteristics ---
Parallelization efficiency      1.0000
Execution time                 57.3396
Processors                           1
Threads amount                       1
Total time                     57.3396
Productive time                57.3396 ( CPU= 56.8142 Sys= 0.5254 I/O= 0.0000 )
--- The comparative characteristics ---
                              Tmin N proc      Tmax N proc      Tmid
Execution time              57.3396     1    57.3396     1    57.3396
User CPU time               56.8142     1    56.8142     1    56.8142
Sys. CPU time                0.5254     1     0.5254     1     0.5254
Processors                        1     1          1     1          1
--- The execution characteristics ---
                                  1
Execution time              57.3396
User CPU time               56.8142
Sys. CPU time                0.5254
Processors                        1
```

The DVMH performance analyzer constructs hierarchical description of a program:

- performance analysis of specific regions of code (a loop nest, iteration of a loop, etc.).
- target regions of code can be chosen manually or automatically set by the compiler according to user provided level of instrumentation.

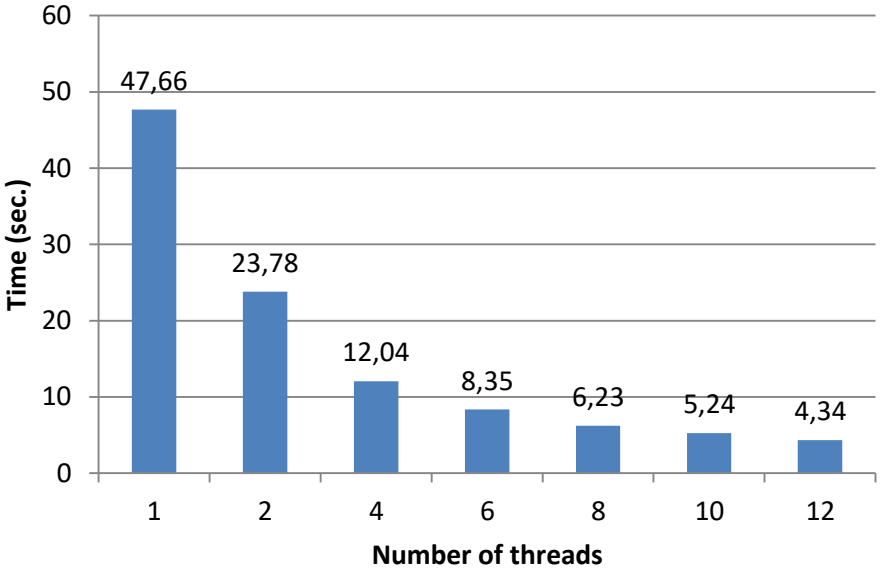
Profiling of all the loops and intervals specified by **INTERVAL** and **END INTERVAL** directives.

```
./dvm f -e4 jacobi.f
./dvm pa sts.gz+ jacobi.peft.txt
```

Step 3.1: incremental parallelization for a multiprocessor

- Incremental parallelization and quick estimation of the possible performance of DVMH parallelization across CPU and GPU cores before full-scale parallelization.
- Possibility to use DVMH parallelization inside the cluster node in MPI programs.

```
      double precision function dostep(f, newf, r, rdx2, rdy2,
     &                                  rdz2, beta, mx, my, mz)
        integer :: mx, my, mz
        double precision, dimension(mx,my,mz) :: f, newf, r
        double precision :: rdx2, rdy2, rdz2, beta, eps
        integer :: i, j, k
        eps = 0.
CDVM$ PARALLEL (k,j,i), REDUCTION(max(eps))
        do k = 2, mz - 1
          do j = 2, my - 1
            do i = 2, mx - 1
              newf(i, j, k) = ((f(i-1,j,k)+f(i+1,j,k))*rdx2
     &                        +(f(i,j-1,k)+f(i,j+1,k))*rdy2
     &                        +(f(i,j,k-1)+f(i,j,k+1))*rdz2
     &                        -r(i,j,k)) * beta
              eps = max(eps,abs(newf(i,j,k)-f(i,j,k))
            enddo
          enddo
        enddo
        dostep = eps
      end function
```



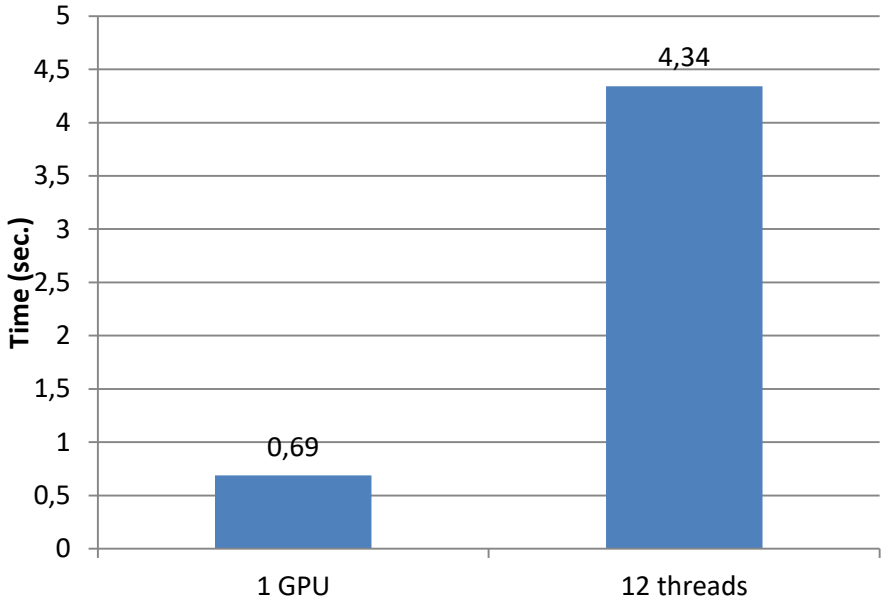
K10 cluster (KIAM RAS):

1 node
- 2 Intel Xeon E5-2660 (8-core)
- 3 NVIDIA Fermi M2090

Step 3.1: incremental parallelization for a GPU

- Incremental parallelization and quick estimation of the possible performance of DVMH parallelization across CPU and GPU cores before full-scale parallelization.
- Possibility to use DVMH parallelization inside the cluster node in MPI programs.

```
      double precision function dostep(f, newf, r, rdx2, rdy2,
     &                                 rdz2, beta, mx, my, mz)
        integer :: mx, my, mz
        double precision, dimension(mx,my,mz) :: f, newf, r
        double precision :: rdx2, rdy2, rdz2, beta, eps
        integer :: i, j, k
CDVM$ ACTUAL(eps)
        eps = 0.
CDVM$ REGION INOUT(f,newf, eps), IN(r,rdx2,rdy2,rdz2,beta)
CDVM$ PARALLEL (k,j,i), REDUCTION(max(eps))
        do k = 2, mz - 1
          do j = 2, my - 1
            do i = 2, mx - 1
              newf(i, j, k) = ((f(i-1,j,k)+f(i+1,j,k))*rdx2
     &                        +(f(i,j-1,k)+f(i,j+1,k))*rdy2
     &                        +(f(i,j,k-1)+f(i,j,k+1))*rdz2
     &                        -r(i,j,k)) * beta
              eps = max(eps,abs(newf(i,j,k)-f(i,j,k))
            enddo
          enddo
        enddo
CDVM$ ENDREGION
CDVM$ GET_ACTUAL(eps)
        dostep = eps
      end function
```



K10 cluster (KIAM RAS):

1 node
- 2 Intel Xeon E5-2660 (8-core)
- 3 NVIDIA Fermi M2090

Step 3.2: incremental parallelization for a cluster

Incremental parallelization based on insertion of temporary arrays can be obtained with only local program transformations affected the time-consuming regions we are interested in.

In spite of it still may degrade the entire program performance it allows us to:

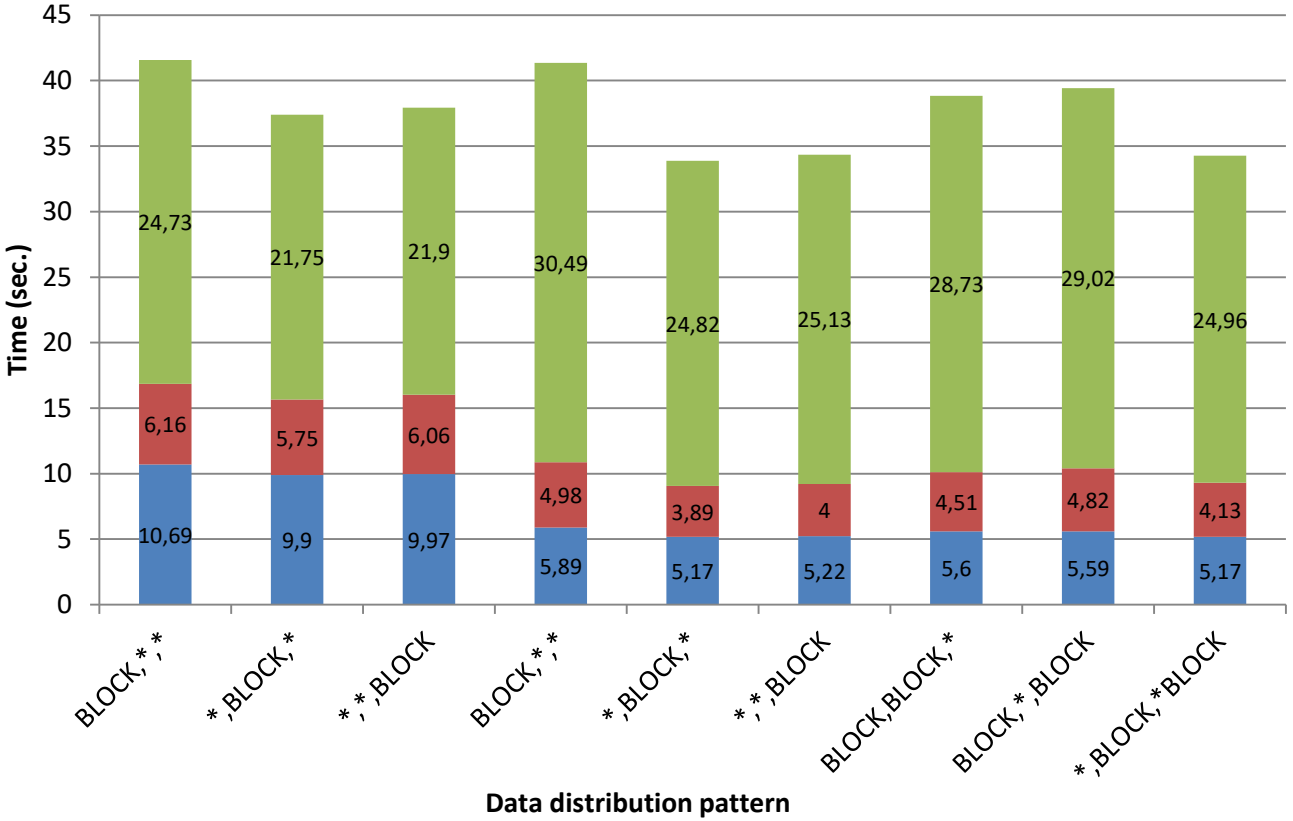
- estimate the possible performance gain each time-consuming source code region may separately achieve if data and computation distribution is accomplished in the way that is most suitable to the code region,
- estimate the variety of desirable data and computation distributions for the number of code regions the program contains.

```
  double precision function dostep(f, newf, r, rdx2, rdy2,
     &                                     rdz2, beta, mx, my, mz)
        integer :: mx, my, mz
        double precision, dimension(mx,my,mz) :: f, newf, r
        double precision :: rdx2, rdy2, rdz2, beta, eps
        integer :: i, j, k
C      AUXILARY TEMPORARY ARRAYS
        double precision, dimension(mx,my,mz) :: f_, newf_, r_
C      DATA DISTRIBUTION
CDVM$ DISTRIBUTE (BLOCK,BLOCK,BLOCK) :: f
CDVM$ ALIGN newf(i,j,k) WITH f(i,j,k)
CDVM$ ALIGN r(i,j,k) WITH f(i,j,k)
CDVM$ ACTUAL(eps)
        eps = 0.

CDVM$ INTERVAL(1)
C      COPY BEFORE
        f = f_
        newf = newf_
        r = r_
CDVM$ END INTERVAL
```

```
CDVM$ REGION
CDVM$ PARALLEL (k,j,i) ON newf(i,j,k), REDUCTION(max(eps)),
CDVM$* SHADOW_RENEW(f)
        do k = 2, mz - 1
          do j = 2, my - 1
            do i = 2, mx - 1
              newf(i, j, k) = ((f(i-1,j,k)+f(i+1,j,k))*rdx2
     &                        +(f(i,j-1,k)+f(i,j+1,k))*rdy2
     &                        +(f(i,j,k-1)+f(i,j,k+1))*rdz2
     &                        -r(i,j,k)) * beta
              eps = max(eps,abs(newf(i,j,k)-f(i,j,k))
            enddo
          enddo
        enddo
CDVM$ ENDREGION
CDVM$ GET_ACTUAL(eps)
CDVM$ INTERVAL(3)
C      COPY AFTER
        f_ = f
        newf_ = newf
        r_ = r
CDVM$ END INTERVAL
        dostep = eps
      end function
```

Step 3.3: incremental parallelization for a cluster



K10 cluster (KIAM RAS):

1 node
- 2 Intel Xeon E5-2660 (8-core)
- 3 NVIDIA Fermi M2090

Copy after

Parrallel loop

Copy before

Step 4: full-scale parallelization for a heterogeneous cluster

- Final fusion of the desirable data and computation distributions in a single one should take into account their performance impact estimated on the previous step.
- However, this fusion may require a significant transformation of the source code which is not covered by code regions parallelized on previous steps.

```
 program jacobi
       double precision, allocatable, dimension(:,:,:) :: f, newf
       double precision, allocatable, dimension(:,:,:) :: r
CDVM$ DISTRIBUTE :: f
CDVM$ ALIGN :: newf, r
       ...
       allocate(f(mx, my, mz))
       allocate(newf(mx, my, mz))
       allocate(r(mx, my, mz))
CDVM$ REDISTRIBUTE (BLOCK, BLOCK, BLOCK) :: f
CDVM$ REALIGN (i,j,k) WITH f(i,j,k) :: newf, r
       curf = 0
       do n = 1, NITER
         if (curf .eq. 0) then
           eps = dostep(f, newf, r, rdx2, rdy2, rdz2,
     &                  beta, mx, my, mz)
         else
           eps = dostep(newf, f, r, rdx2, rdy2, rdz2,
     &                  beta, mx, my, mz)
         endif
         print *, 'Iteration=' , n, 'eps=', eps
         curf = 1 - curf
         enddo
      end
```

```
      double precision function dostep(f, newf, r, rdx2, rdy2,
     &                                 rdz2, beta, mx, my, mz)
         integer :: mx, my, mz
         double precision, dimension(mx,my,mz) :: f, newf, r
         double precision :: rdx2, rdy2, rdz2, beta, eps
         integer :: i, j, k
CDVM$ INHERIT f,newf,r
CDVM$ ACTUAL(eps)
         eps = 0.
CDVM$ REGION
CDVM$ PARALLEL (k,j,i) ON newf(i,j,k), REDUCTION(max(eps)),
CDVM$* SHADOW_RENEW(f)

         do k = 2, mz - 1
           do j = 2, my - 1
             do i = 2, mx - 1
               newf(i, j, k) = ((f(i-1,j,k)+f(i+1,j,k))*rdx2
     &                         +(f(i,j-1,k)+f(i,j+1,k))*rdy2
     &                         +(f(i,j,k-1)+f(i,j,k+1))*rdz2
     &                         -r(i,j,k)) * beta
             eps = max(eps,abs(newf(i,j,k)-f(i,j,k))
             enddo
           enddo
         enddo
CDVM$ ENDREGION
CDVM$ GET_ACTUAL(eps)
         dostep = eps
       end function
```

Step 5: tuning parallel program performance

The factors which influence parallel program performance on a distributed memory system:

- program parallelism which is a part of parallel calculations in total volume of calculations,
- load balancing of processors,
- time of interprocessor communications,
- degree of overlapping of interprocessor communications with calculations.

DVMH runtime knowns:

- whether sequential or parallel part of the program is executed on any processor at any moment,
- all synchronization and communication points,
- reason for communication (exchange of shadow edges, access to remote data, reduction operations, etc.).

During a program execution the DVMH runtime stores time characteristic information in processor memory and writes the data into a file upon the program completion.

The performance visualizer allows the user to get time characteristics of the program execution in more or less detail.

Main characteristics and their components

```
Processor system=4*2*1
Statistics has been accumulated on DVM-system version 5.0, platform K10
Analyzer is executing on DVM-system version 5.0, platform K10
-------------------------------------------------------------------------------
INTERVAL ( NLINE=17 SOURCE=bt.fdv ) LEVEL=0  EXE_COUNT=1
--- The main characteristics ---
Parallelization efficiency      0.9091
Execution time                131.8853
Processors                           8
Threads amount                       8
Total time                   1055.0828
Productive time               959.1438 ( CPU= 956.9820 Sys= 2.1597 I/O= 0.0021 )
Lost  time                     95.9390
   Insufficient parallelism    22.2913 ( User= 7.1590 Sys= 15.1324 )
   Communication               73.5646 ( Real_sync= 0.0000 Starts= 0.0000 )
   Idle time                    0.0830
Load imbalance                 33.0604
Synchronization                23.0580
Time variation                 21.5797
Overlap                         0.0047
Productive time GPU           858.6074
Lost time GPU                  63.0325
                  Nop   Communic    Synchro  Variation    Overlap
I/O                42     0.0000     0.3958     0.3955     0.0000
Reduction           2     0.0277     0.0116     0.0116     0.0000
Shadow           1019    73.4882    22.6506    21.1725     0.0047
--- The comparative characteristics ---
                          Tmin N proc       Tmax N proc       Tmid
Lost time                 7.8598     6      16.5130     1     11.9924
User insufficient par.    0.6583     7       1.1449     4      0.8949
Sys.insufficient par.     1.1182     1       2.8811     4      1.8915
Idle time                 0.0000     1       0.0215     6      0.0104
Communication             3.8937     4      14.7201     1      9.1956
Synchronization           0.1995     3       4.4655     6      2.8822
Variation                 0.4122     3       4.2150     6      2.6975
Overlap                   0.0000     1       0.0042     3      0.0006
Load imbalance            0.0000     6       8.6532     1      4.1326
Execution time          131.8638     6     131.8853     1    131.8750
User CPU time           115.2126     1     123.6255     6    119.6227
Sys. CPU time             0.1595     1       0.4113     4      0.2700
```

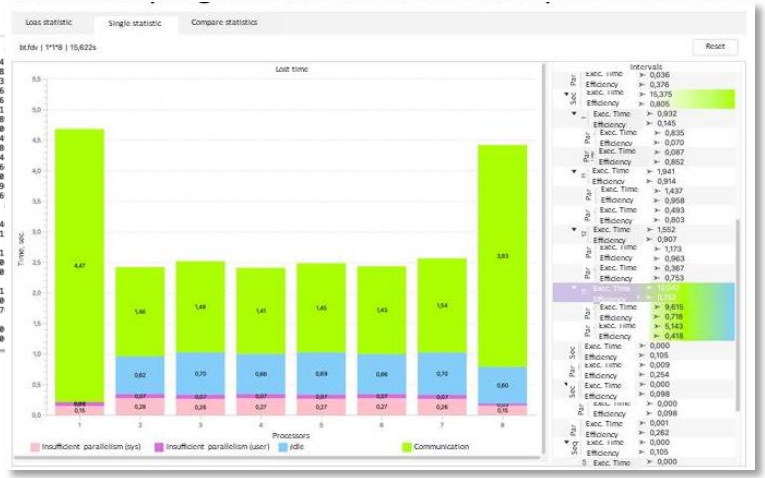
Main characteristics and their components

- Time of the program execution (**Execution time**).
- The number of used processors (**Processors**).
- Total processor time (**Total time**) = **Execution time * Processors**.
- Productive time (**Productive time**) predicted execution time on a single processor.
- Efficiency coefficient (**Parallelization efficiency**) = **Productive time / Total time**
- Lost time (**Lost time**) = **Total time - Productive time.**
- Possible reasons which produce the lost time.

```
--- The execution characteristics ---
                              1          2          3          4          5          6
Lost time              16.5130    15.0269     9.9889     7.9239     9.8276     7.8598    15.184
User insufficient par.  0.6747     0.7012     1.0742     1.1449     1.0838     1.1333     0.658
Sys.insufficient par.   1.1182     1.5641     2.1259     2.8811     2.0773     2.8001     1.163
Idle time               0.0000     0.0131     0.0142     0.0042     0.0095     0.0215     0.006
Communication          14.7201    12.7486     6.7746     3.8937     6.6571     3.9048    13.356
Synchronization         3.8128     4.0006     0.1995     1.1890     3.5263     4.4655     2.771
Variation               2.7274     3.5188     0.4122     1.2950     3.3836     4.2150     2.548
Overlap                 0.0000     0.0000     0.0042     0.0004     0.0000     0.0000     0.000
Load imbalance          8.6532     7.1671     2.1290     0.0641     1.9677     0.0000     7.324
Execution time        131.8853   131.8723   131.8712   131.8811   131.8759   131.8638   131.878
User CPU time         115.2126   116.6350   121.5928   123.5498   121.7610   123.6255   116.534
                                                                             0.3998     0.166
                                                                             0.0003     0.000
                                                                           108.7052   106.339
                                                                            10.1046     5.976
                                                                                  8
                                                                             0.0025     0.004
                                                                             3.8537    13.351
                                                                             0.1118     0.021
                                                                             0.0040     0.000
                                                                             4.3497     2.750
                                                                             0.1118     0.021
                                                                             0.0040     0.000
                                                                             4.0992     2.527
                                                                             0.0000     0.000
                                                                             0.0000     0.000
```

```
--- The GPU characteristics ---
Proc: #1
  GPU #1 (Tesla M2090)
                                  #        Min        Max        Sum    Average  Productive       Lost
[Shadow] Copy GPU to CPU       112568    1.602K     2.441M     1.163G    10.831K           -     4.3247s
[Shadow] Copy CPU to GPU        19921    6.258K     1.807M   459.572M    23.623K           -     0.7563s
[Region IN] Copy CPU to GPU         8        12B    21.160M    43.837M     5.480M    0.0083s           -
Loop execution                  76481     0.0003     0.8388   105.572S     0.0014   105.5728s           -
Data reorganization               811    21.160M    22.192M    17.167G    21.675M           -     0.4827s
Reduction                          10     0.0000     0.0004     0.0006     0.0001           -     0.0006s
Productive time:   105.5811s
Lost time      :     5.5642s

Proc: #2
  GPU #2 (Tesla M2090)
                                  #        Min        Max        Sum    Average  Productive       Lost
[Shadow] Copy GPU to CPU        71362    3.125K     2.441M   952.859M    13.673K           -     3.1784s
[Shadow] Copy CPU to GPU        61127    1.562K     1.259M   715.215M    11.980K           -     2.0936s
[Region IN] Copy CPU to GPU         8        12B    21.160M    43.850M     5.481M    0.0083s           -
Loop execution                  76481     0.0003     0.8413   106.184Z     0.0014   106.1842s           -
Data reorganization              1013    21.160M   105.799M    38.248G    38.664M           -     1.0437s
Reduction                          10     0.0000     0.0004     0.0006     0.0001           -     0.0006s
Productive time:   106.1925s
Lost time      :     6.3164s
```



Characteristics of program execution on each processor or GPU

Dynamic tuning methods for DVMH programs

- Data and computation distribution between cluster nodes according to their performance.
- Data and computation distribution between GPUs and CPU cores according to their performance:
  - ✓ simple static mode – data and computation distribution in accordance with a user specified weights,
  - ✓ simple dynamic mode – data and computation distribution is selected during a program execution,
  - ✓ dynamic with selection mode – profile-guided data and computation distribution.
- Data transformation at runtime to choose the right memory access pattern.
- Dynamic CUDA handler compilation during the program execution.

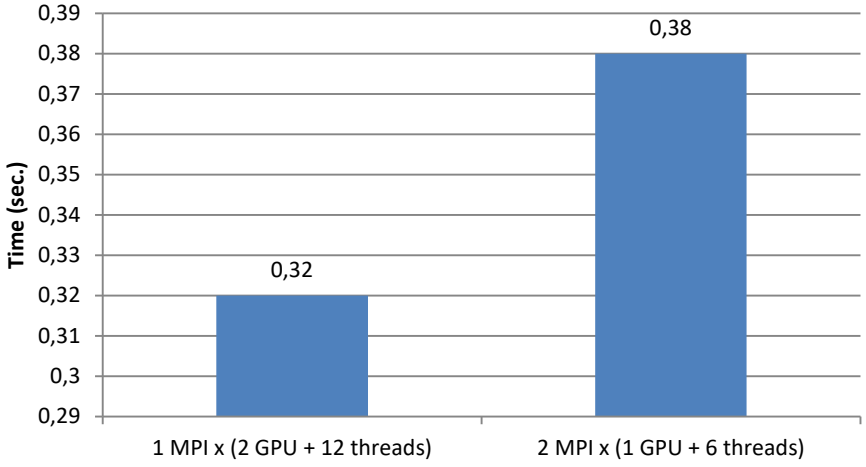
More than 20 environment variables to control program execution:

```
export DVMH_PPN='2,1,1'              # Number of process per node
export DVMH_NUM_THREADS='8,240,240'  # Number of CPU threads per process
export DVMH_NUM_CUDAS='3'            # Number of GPUs per process

export DVMH_CPU_PERF=''              # Performance of all cores of CPU per process
export DVMH_CUDAS_PERF=''            # Performance of each GPU per device
export DVMH_SCHED_TECH='dynamic1'    #  Schedule mode

export DVMH_SET_AFFINITY='enable'    # Thread affinity control
export DVMH_NO_DIRECT_COPY='1'       # Don't use GPUDirect transfers
export DVMH_IO_BUF_SIZE='10485760'   # Size of input/output buffer
...
```

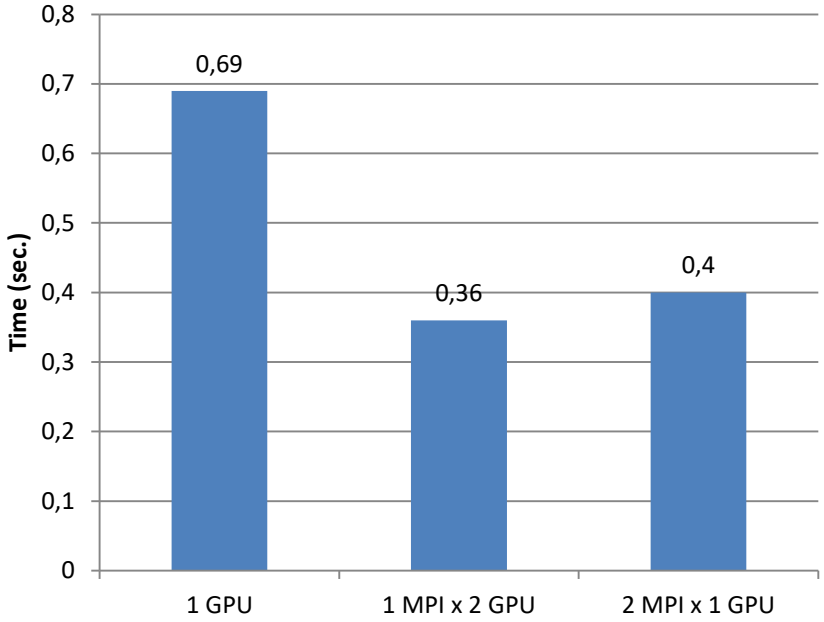
Comparison of execution time on a multiprocessor and GPU



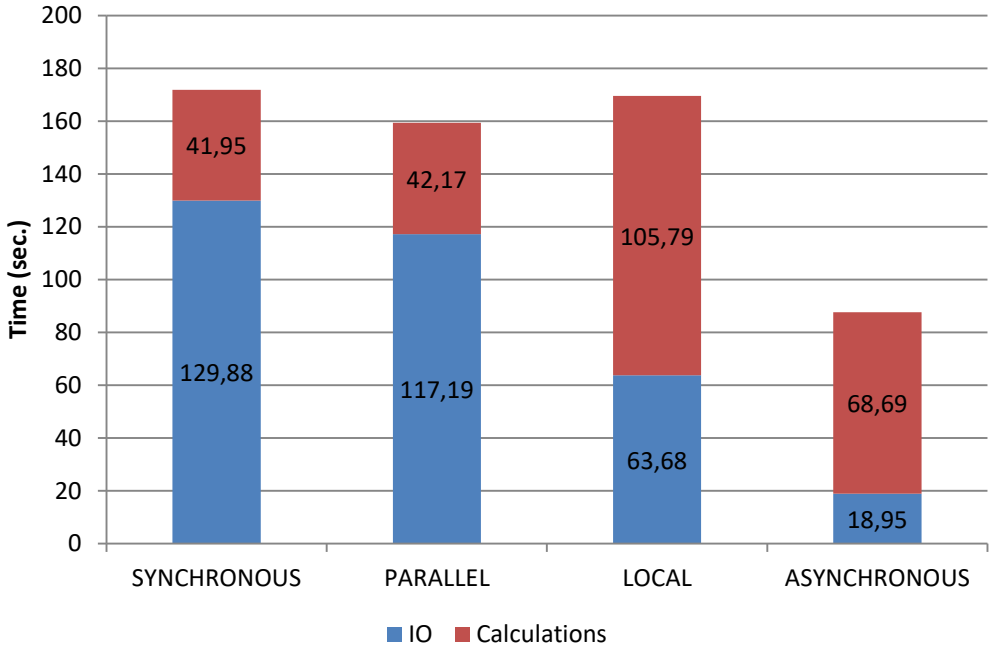
Static distribution of data and computation between GPUs and CPU cores according two weights estimated at preliminary parallel program profiling.

K10 cluster (KIAM RAS):

1 node { 2 Intel Xeon E5-2660 (8-core)
3 NVIDIA Fermi M2090



Step 6: program reliability and parallel IO in the DVMH model




```
//      C/C++
const char *mode = "wb";
FILE *cp = fopen("jac_%02d.dat", mode);

!       Fortran
!DVM$  IO_MODE (LOCAL, ASYNC)
       open(4, ACCESS='STREAM', FILE='DATA_%02d.DAT', ERR=44)
       ...
       write(4) A(2:L-1,2:L-1), B
       ...
       close(4)
```

Parameter	IO mode
“wb”	Synchronous serial I/O
“wbl”	Synchronous parallel I/O to a local file
“wbp”	Synchronous parallel I/O to a parallel file
“wbs”	Asynchronous serial I/O
“wbsl”	Asynchronous parallel I/O to a local file
“wbsp”	Asynchronous parallel I/O to a parallel file

Lomonosov (RCC MSU):

1 node
- 2 Intel Xeon 5570/5670 (4/6-core)
- 1 NVIDIA X2070

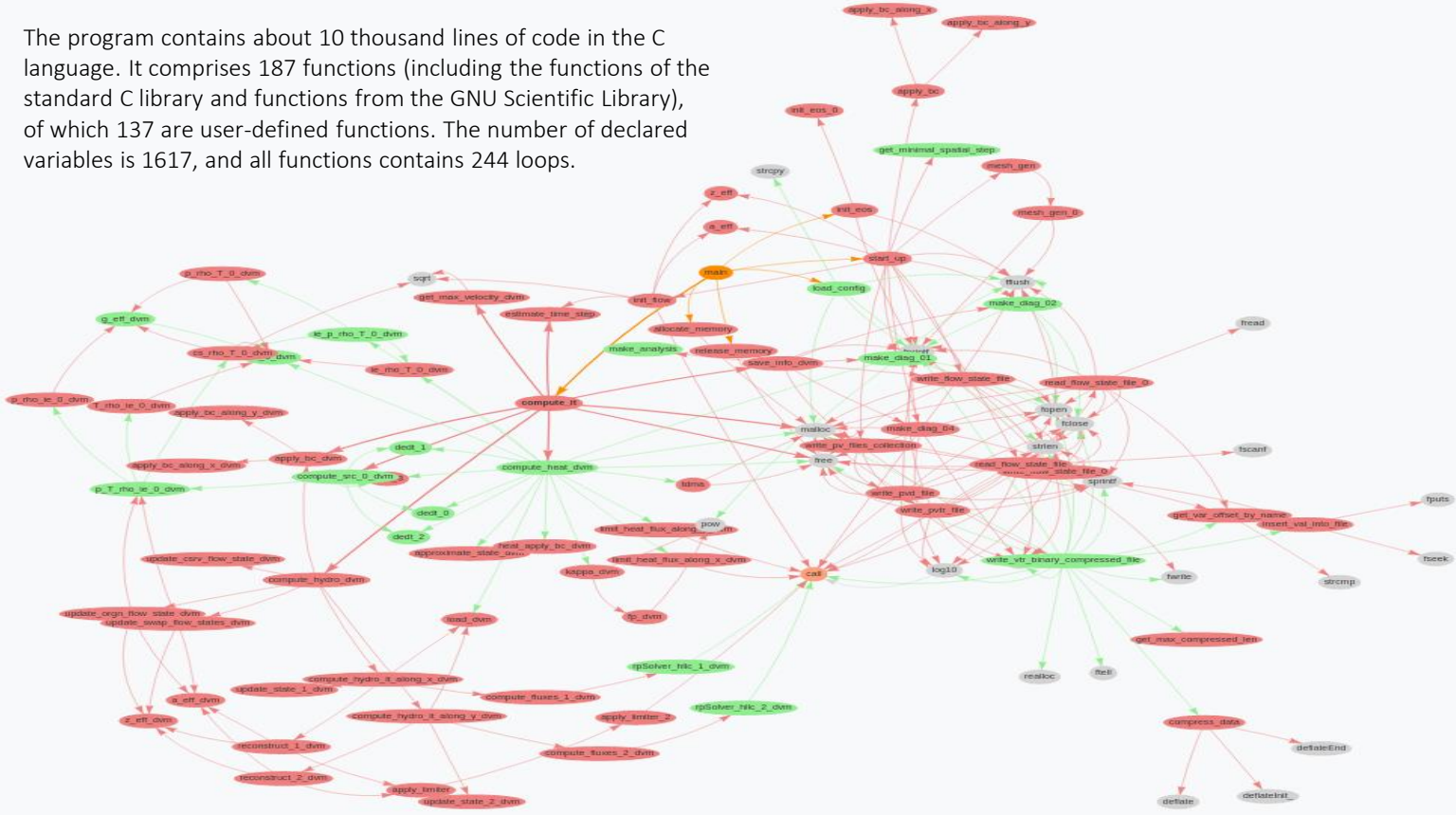
Method: Jacoby
Grid size: 32000 x 32000
Number of iterations: 100
Number of checkpoints: 10

History of success

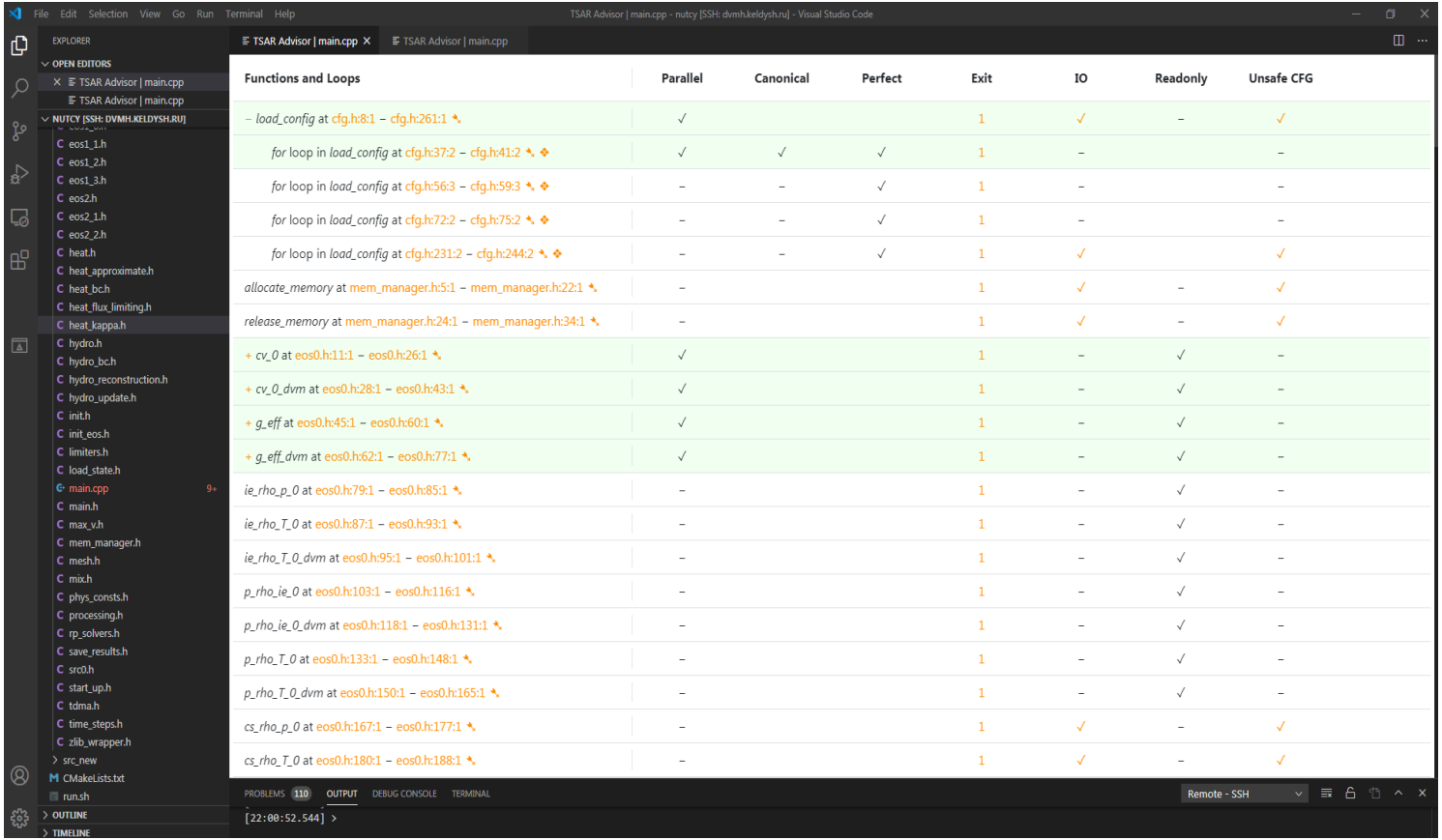
- REACTOR – neutron physical design of nuclear reactors and hybrid nuclear facilities,
- Elasticity3D – numerical simulation of 3D seismic fields in elastic media with complex geometry of the free surface,
- HyperbolicSolver2D – solution of systems of hyperbolic equations in 2D domains of complex shape using an explicit and unstructured meshes,
- GIMM_APP_Powder_3D/GIMM_APP_Powder_3D – parallel programs for 2D and 3D simulation of melting multicomponent powders under the influence of selective laser sintering based on the multicomponent and multiphase fluid dynamics model,
- QuantumBitStates – calculation of states of quantum computer cubits by solving the unsteady Schrödinger equation for two particles taking into account their spins,
- GIMM_APP_Crystal_2D/GIMM_APP_Crystal_3D – parallel programs for 2D and 3D simulation of 3D crystallization processes under the influence of laser or electron beam based on the multicomponent and multiphase fluid dynamics model,
- ThermalConductivity – solution of the boundary value problem for the two-dimensional quasilinear parabolic equation written in conservation form in various statements on unstructured triangular meshes,
- MHPDV – simulation of the spherical explosion in an external magnetic field by solving the equations of perfect magnetic hydrodynamics,
- NCOM – simulation of multicomponent multiphase isothermal filtering for oil and gas deposits,
- Cavity – simulation of circulatory flow in a planar square cavity with a moving upper lid,
- Container – simulation of flow of heavy viscous fluid under the gravity force in a rectangular container with an open upper wall and a hole in a lateral wall,
- and other.

A software packaged for numerical simulation of hydrodynamic instabilities

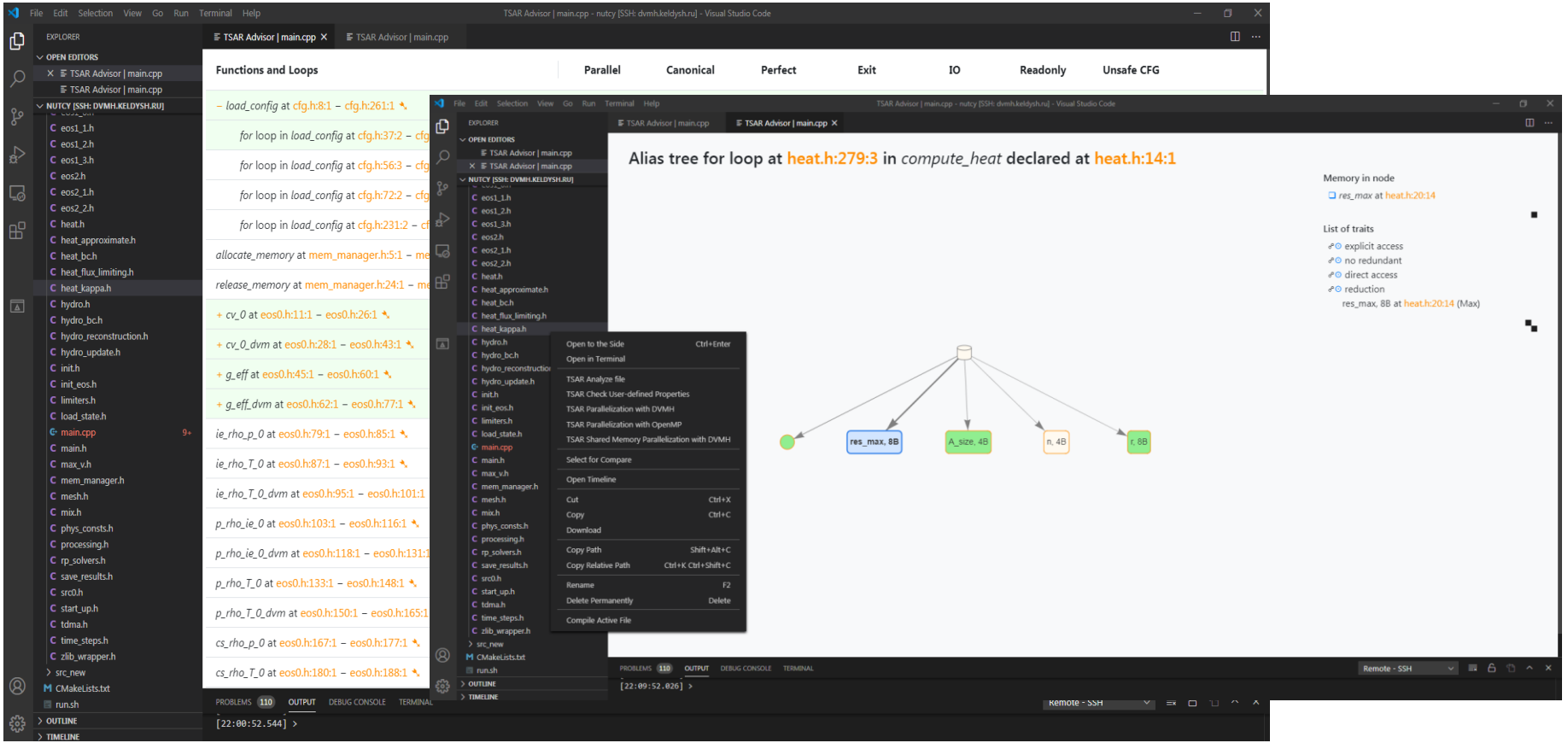
The program contains about 10 thousand lines of code in the C language. It comprises 187 functions (including the functions of the standard C library and functions from the GNU Scientific Library), of which 137 are user-defined functions. The number of declared variables is 1617, and all functions contains 244 loops.



Interactive assistance tool: program structure and loop properties

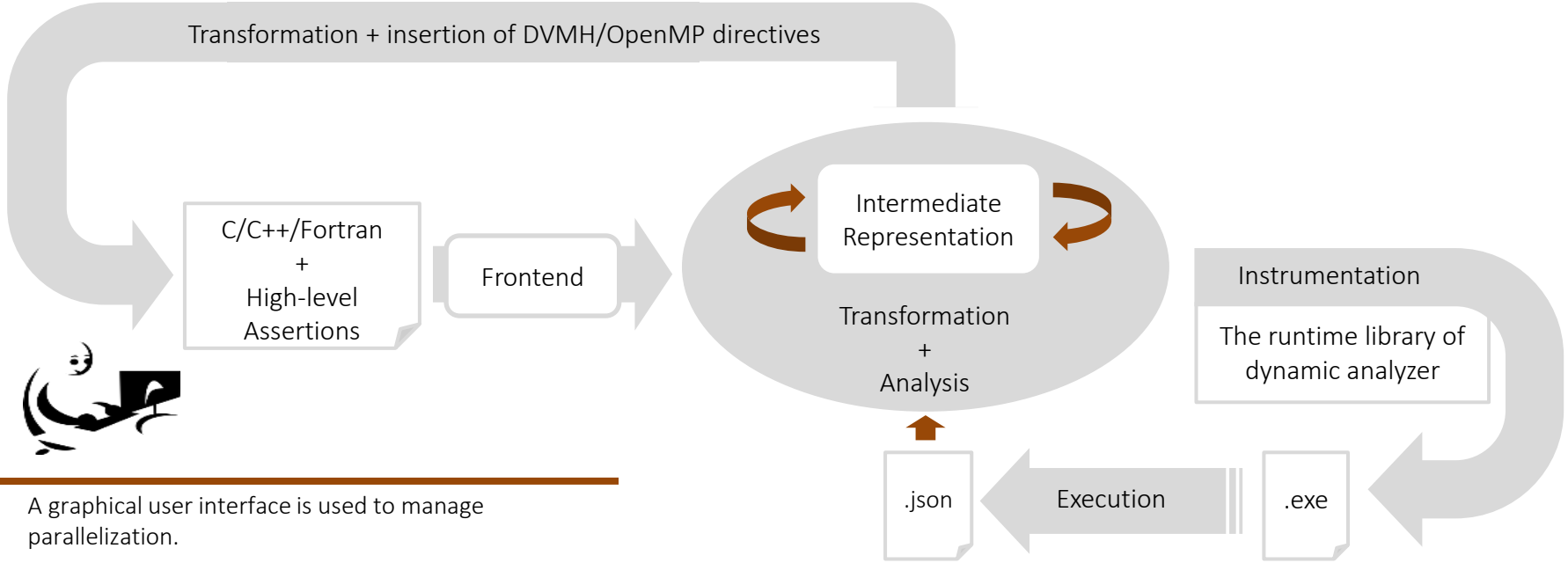


Interactive assistance tools: alias analysis and data dependencies



Architecture of SAPFOR

SAPFOR (System For Automate Parallelization) is a software development suit that is focused on cost reduction of manual program parallelization.



- A graphical user interface is used to manage parallelization.
- Build automation tools, such as Make, can be also used to run program analysis.

Automated program transformation

Different array parameters of a function refers to different physical quantities. Each element of the array represents a compute state in a corresponding grid point. Thus, different array parameters point to different memory locations and do not alias.

However, the C language turns accesses to arrays of pointers into sequences of two dereference statements that inhibits accurate alias analysis.

A demand-driven source-to-source transformation in SAPFOR aimed at splitting small arrays of pointers into independent variables. Each element of the original array-parameter results in an independent parameter of a pointer type, hereafter we can apply the *restrict* qualifier.

1. Automated replacement of an array-parameter in a function:

```
void foo(state_t **ss) {
#pragma spf transform replace(ss) nostrict
  /* accesses to ss[0] and ss[1] */
}
```

⇨

```
/* Replacement for void foo(state_t **ss) */
void foo_spf0(state_t *ss_0, state_t *ss_1) {
#pragma spf transform metadata \
            replace(foo, { .0 = ss_0, .1 = ss_1})
  /* accesses to ss_0 and ss_1 */
}
```

2. Automated replacement of the calling function:

```
void bar(state_t **ss) {
#pragma spf transform replace with(foo_spf0)
  foo(ss);
}
```

⇨

```
void bar() {
  foo_spf0(ss[0], ss[1]);
}
```

Program parallelization summary

The original and resulting programs total 10000 and 21450 lines of code correspondingly.

Static (1 min. 42 sec.) ana dynamic (6 min. 49 sec., slowdown 2045 times) analysis were applied

Manual transformations were applied to enable data partitioning and offloading computation to GPU:

- array delinearization,
- replacement of indirect function calls with direct ones,
- replacement of calls to GSL library functions with a manually written code

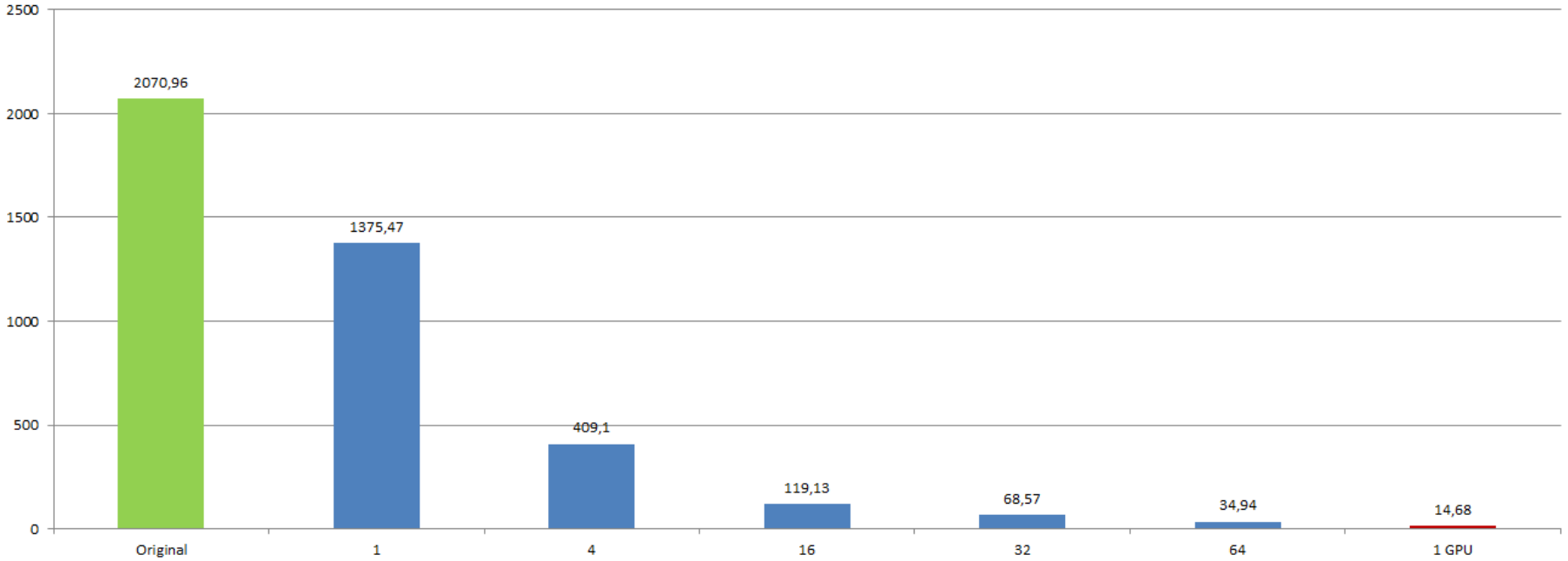
Semi-automatic transformations were applied to break data dependencies, to obtain perfectly nested loops, to increases static analysis accuracy:

- array of structures replacement,
- function inlinining,
- array expansion,
- loop distribution.

The DVMH specifications were inserted (500 lines of code):

- 107 directives to specify parallel loop nests (`parallel`),
- 20 directives to specify execution on GPU (`region`, `get_actual`, `actual`),
- 160 data distribution directives (`distribute`, `align`, `realign`, `redistribute`),
- 66 directives to specify function that inherit data distribution from caller function (`inherit`),
- 5 directives to specify access to remote data (`remote_access`).

The execution time (sec.) of 100 iterations on the grid 3000x1528



K60 cluster (KIAM RAS):

1 node { 2 Intel Xeon Gold 6142v4 (16-core)
4 NVIDIA V100 GPU

Performance profiling on 1 and 4 GPUs (grid size 12000x5028)

GPUs improve the program performance and reduces the execution time from 162 second on 1 GPU to 48 second on 4 GPUs (3.3 times).

1 GPU (Tesla V100-PCIE-32GB)

	#	Min	Max	Sum	Average	Productive	Lost
[Region IN] Copy CPU to GPU	3146	4B	1.350G	25.205G	8.204M	2.2477s	-
Loop execution	5600	0.0000	0.2657	158.7707	0.0284	158.7707s	-
Reduction	300	0.0006	0.0013	0.1837	0.0006	-	0.1837s
Page lock host memory	6200	0.0000	0.0029	0.7681	0.0001	-	0.7681s

Productive time: 161.0184s
Lost time : 0.9517s

4GPU # (Tesla V100-PCIE-32GB)

	#	Min	Max	Sum	Average	Productive	Lost
[Shadow] Copy GPU to CPU	9620	19.664K	140.672K	416.625M	44.347K	-	0.5162s
[Shadow] Copy CPU to GPU	7386	19.664K	281.344K	513.096M	71.136K	-	0.4169s
[Region IN] Copy CPU to GPU	3214	4B	346.047M	6.308G	2.010M	1.1567s	-
GET_ACTUAL	110	115.349M	346.047M	16.897G	157.294M	2.7507s	-
Loop execution	4200	0.0000	0.0607	38.9186	0.0093	38.9186s	-
Reduction	300	0.0002	0.0008	0.0565	0.0002	-	0.0565s
Page lock host memory	6294	0.0000	0.1056	4.6534	0.0007	-	4.6534s

Productive time: 42.8260s
Lost time : 5.6431s

Conclusion

Directive-based programming models help the programmer to accomplish concerns of parallel programming from the complexity, correctness, and portability and maintainability perspectives.

High-level programming model should be:

- general enough to solve tasks in a wide domain,
- wide and extendable enough to support different parallel architectures,
- implicit enough to hide from the application programmer implementation details,
- explicit enough to allow the compiler to optimize programs for a chosen architectures.

It is feasible to support multiple levels of parallelization in a single parallel programming model.

DVMH programs can be executed without any changes on workstations and HPC systems equipped with multicore CPUs, GPUs, and Intel Xeon Phi coprocessors.

The performance gains, which are achieved on different architectures, are caused by various optimizations implemented in the DVMH compiler and runtime system.

At startup the programmer configures desirable resources (the number of cluster nodes, threads and accelerators, the number of processors per node as well as performance of different processing units) the parallel application should utilize.
Thus the best configuration can be selected to improve the efficiency of computational resources utilization in HPC centers.

Development of assistant tools and automated parallelization techniques on top of a high-level programming model may further reduce the effort required to embed parallelism into the existing application programs.

Thank you for your attention

URL

http://dvm-system.org

dvm@keldysh.ru

E-mail

PaCT-2023